

Courtesy of  
<https://mlvu.github.io>

offline machine learning: the basic recipe

Lecture 5:  
Abstract (part of) your problem *Prepare your data*  
Classification, Regression, Clustering, Density estimation, Generative Modeling, Online learning, Reinforcement Learning, Structured Output Learning

Choose your **instances** and their **features**.  
For supervised learning, choose a target.

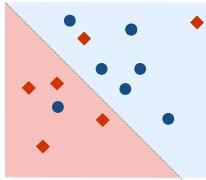
Choose your **model class**.  
Linear models, Decision Trees, kNN,

Search for a good model.  
Usually, a model comes with its own search method. Sometimes multiple options are available.

today:  
*Evaluate your model*

Here is the basic recipe for machine learning again. This week, we'll discuss what happens before and after. Today: once you've trained some models, how do you figure out which of them is best?

binary classification



- Positive class
- ◆ Negative class

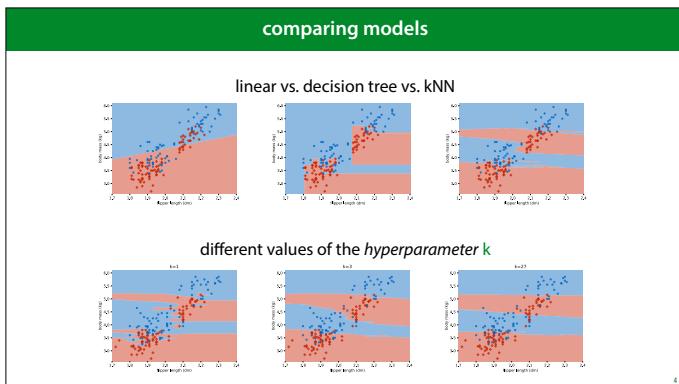
The classifier is a detector for the **positive** class.

error: 3/14  
proportion of misclassifications

accuracy: 11/14  
proportion of correct classifications

We'll focus mostly on **binary classification** today (two-class classification). In this case, we can think of the classifier as a detector for one of the classes (like spam, or a disease). We tend to call this class positive. As in "testing positive for a disease."

In classification, the main metric of performance is the proportion of misclassified examples (which we've already seen). This is called the **error**. The proportion of correctly classified examples is called the **accuracy**.

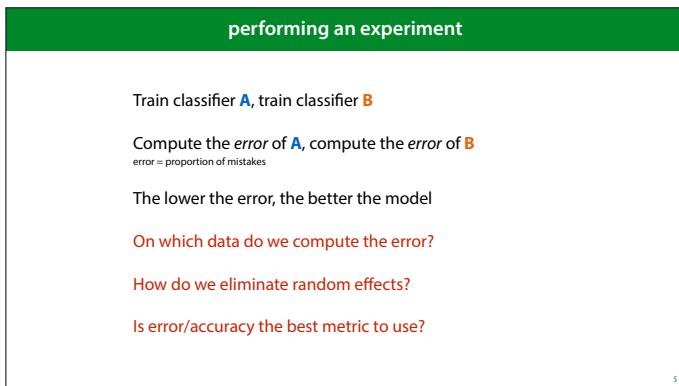


You compare models to figure out which is the best. Ultimately, to choose which model you want to use in production.

*This could be literally the production version of a piece of software, or just the model whose predictions you decide to use in the future.*

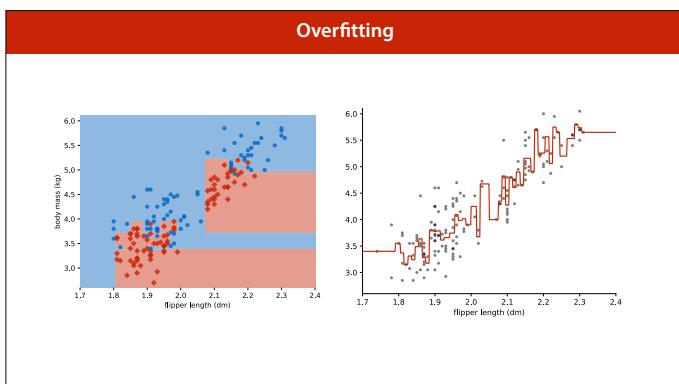
Sometimes you are comparing different model *types* (a decision tree vs a linear model), but you might also be comparing different ways of configuring the same model type. For instance in the kNN classifier, how many neighbours ( $k$ ) should we look at to determine our classification?

With the 2D dataset, we can look at the decision boundary, and make a visual judgment. Usually, that's not the case: our feature space will have hundreds of dimensions, and we'll need to *measure* the performance of a model.



Here is the simplest, most straightforward way to compare two classifiers. You just train them both, so see how many examples they get wrong, and pick the one that made fewest mistakes. This is a very simple approach, but it's basically what we do.

We just need to consider **a few questions**, to make sure that we can trust our results.



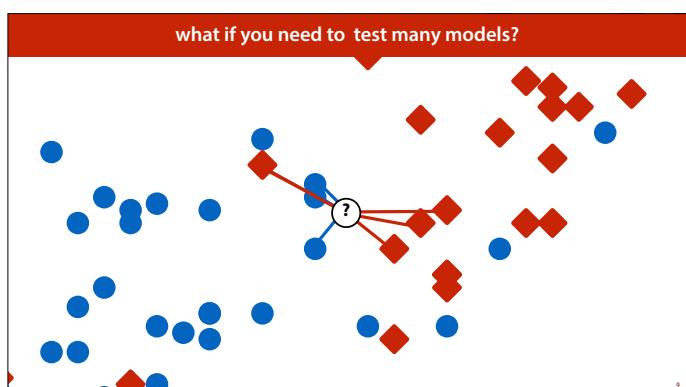
We've already seen what happens when you evaluate on the **training data**. A model that fits the training data perfectly may not be much use when it comes to data you haven't seen before.

# Never judge your performance on the training data

the test set	
training data	test data
The proportion is not important, the <b>absolute size</b> of the test data is.	
We should aim to have at least 500 examples in the test data (10 000 or more is ideal).	

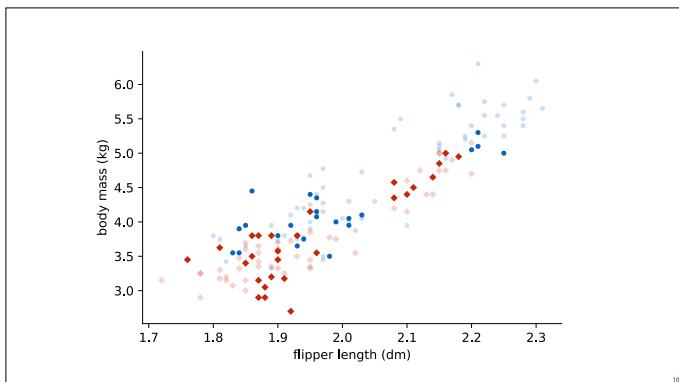
So the first thing we do in machine learning is **withhold** some data. We train our classifiers on the **training data** and test on the **test data**. That way, if we get good performance, we know that we're likely to get a good performance on future data as well, and we haven't just memorised random fluctuations in the training data.

How should we split our data? The most important factor is the size in instances of the **test data**. The bigger this number, the more precise our estimate of our model's error. Ideally, we separate 10 000 test instances, and use whatever we have left over as training data. Unfortunately, this is not always realistic. We'll look at this a little more later.

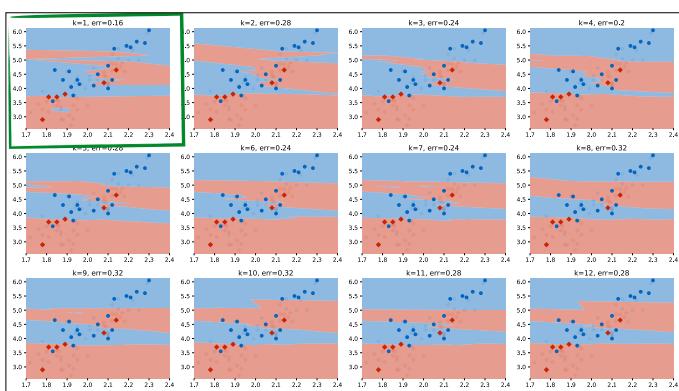


But even if we withhold some test data, we can still go wrong. We'll use k nearest neighbours (kNN) as a running example. Remember, kNN assigns the class of the  $k$  nearest points.

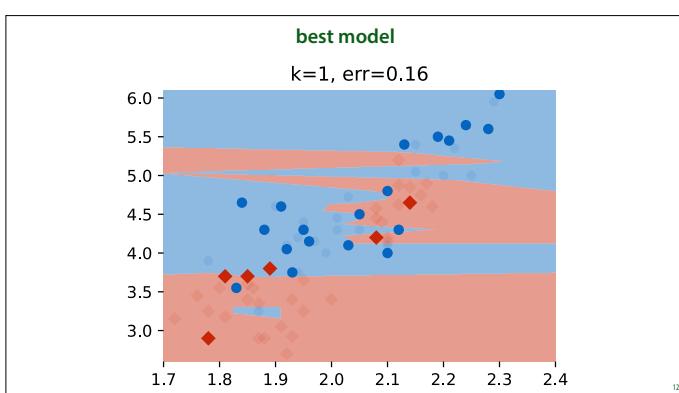
$k$  is what is called a **hyperparameter**. We need to choose its value in some way before we run the algorithm. The algorithm doesn't specify how it should be chosen. One way of choosing  $k$  is to try a few values, and to see for which  $k$  we get the best performance.



We will use the data from the first lecture as an example.  
We will take a small subsample of the dataset, so that the effects that we want to illustrate become exaggerated.



Here we've tested 12 different values of  $k$  on the same [test data](#) (using quite a small test set to illustrate the idea). We can see that for  $k=1$ , we get the best performance. We plotted the test data (with the training data in low opacity).

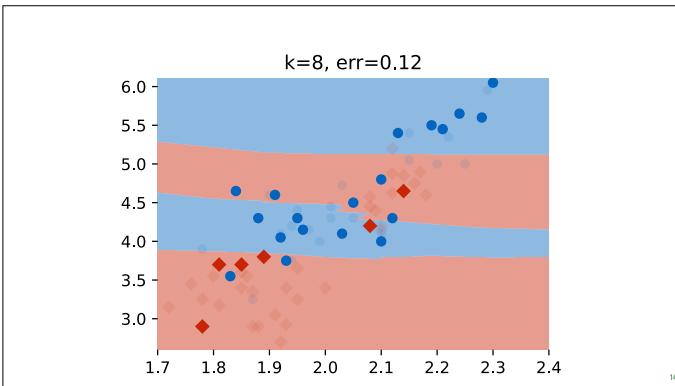


Here is the best run. Should we conclude that  $k=1$  is definitely a better setting than  $k=2$  or  $k=3$ ? Should we conclude that we can expect an error of 0.16 on any future data from the same source?

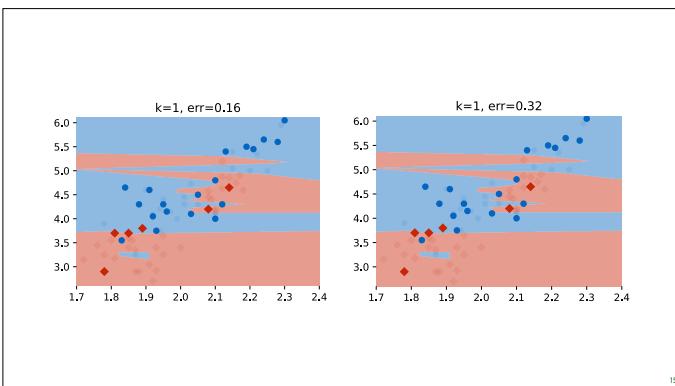


In this case, we have some more data from the same source, so we can evaluate the classifiers again on a **fresh test set**. This is a luxury we don't normally have (we normally use all the data we are given).

What we see is that  $k=1$  no longer gives us the best performance. In fact, we get a radically different best value of  $k$ , and  $k=1$  now gives us the *highest* error in the run.



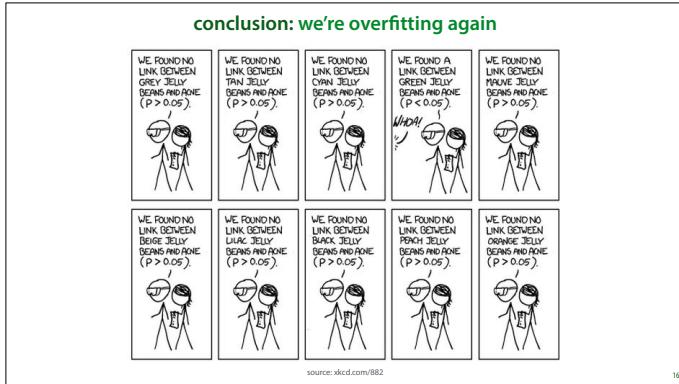
Here is one of the the new best runs.



The same models, different test sets and the conclusions are entirely different. We were diligent in splitting our dataset and evaluating only on withheld data, and yet if we had done only one run on one dataset, as we normally would, we would have concluded that  $k=1$  is the best setting and that an error of 0.16 can be expected with that value.

If we look at the  $k=1$  model from the second run (the one we chose), we will see that the performance on the new test set is terrible. If we select a model in this way and take it into production, we will find that it performs terribly.

So what's happening here?



This is essentially the overfitting problem again. Our method of choosing the hyperparameter  $k$  is just another learning algorithm. By testing so many values of  $k$  on such a small amount of test data, we are overfitting our choice of  $k$  on the test data. The model we choose fits well because of random fluctuations in the data. When we resample the data, these fluctuations disappear and the performance drops.

This is an instance of the *multiple testing* problem in statistics. We're testing so many things, that the likelihood of a noticeable effect popping up by chance increases. We are in danger of ascribing meaning to random fluctuations.

Specifically, in our case, the  $k=1$  classifier **got lucky** on a few examples, that just happened to fall on the right side of the decision boundary. If we draw some new data, the same classifier won't be lucky again. The more different values of  $k$  we try, the more we are in danger of this kind of random luck determining which hyperparameters come out as good.

The simple answer to the problem of multiple testing is **not to test multiple times**.

see also: [https://www.explainxkcd.com/wiki/index.php/882:\\_Significant](https://www.explainxkcd.com/wiki/index.php/882:_Significant)

**evaluation: the modern recipe**

Split your data into **train** and **test** data.  
Sample randomly. At least 500 examples in your test set. In ML benchmarks the test data is often given.

Choose your model, hyperparameters, etc. only using the **training** set.  
Save your test set until the very last minute. Don't use it for anything.

State your hypothesis  
i.e.  $kNN$  with  $k=7$  beats existing model  $X$ , or  $kNN$  with  $k=7$  is better than  $kNN$  with  $k = 12$

Test your hypothesis **once** on the **test** data  
This is usually at the very end of your project when you write your report or paper.

17

There are many different approaches to machine learning experimentation, and not every paper you see will follow this approach, but this is the most common one.

It's important to mention in your paper that you followed this approach, since the reader can't usually see it from the presented results.

## Don't re-use your test data

Just to emphasize the important point: the more you use the test data, the less reliable your conclusions become. Figure out what the end of your project is, and do not touch the test data until the end.

In really important and long-term projects, it's not a bad idea to withhold multiple test sets. This allows you to still test your conclusions in case you've ended up using the original test data too often.

### reusing your test data

- Causes you to **pick the wrong model**
- Inflates your **performance estimate**

Not only does reusing test data mean that you pick the wrong model, it also means that the error estimate you get is probably much lower than the error you would actually get if you gathered some more test data.

### validation set

**training**   **validation**   **test**

During model and hyperparam. selection:

• train on: **training**

• test on: **validation**

Final run:

• train on: **training**   **validation**

• test on: **test**

*← usually*

This means that you need to test which model to use, which hyperparameters to give it, and how to extract your features **only on the training data**. In order not to evaluate on the training data for these evaluations, you usually split the training data **again**: into a (new) **training set** and a **validation set**.

Ideally, your **validation data** is the same size as your **test set**, but you can make it a little smaller to get some more **training data**.

This means that you need to **carefully plan your research process**. If you start out with just a single split and keep testing on the same **test data**, there's no going back (you can't unsee your **test data**). And usually, you don't have the means to gather some new dataset.

It's usually fine in the final run to append the validation data to your training data. This is not always the case however, so if you use a standard benchmark you should check if this is allowed, and if you use your own dataset, you should describe carefully whether you do this.

Note that this approach by itself doesn't *prevent* multiple testing. It just provides for a final failsafe to *detect* it. Before you make the decision to try your model on the test data, you should first convince yourself that the results you see are not down to multiple testing. You can do this by not testing too many hyperparameter values, or if you fear that you have, by rerunning your experiment on a different [train/validation](#) split to double-check.

There's always a bit of a tense moment when you run the experiment on the [test data](#), and you get to find out how close the real numbers you'll get to report are to the numbers you've seen for the [validation](#). However, if your datasets are large, and you haven't done anything strange in the hyperparameter tuning phase, they will usually be very close together.

not this			
	dataset 1	dataset 2	dataset 3
other method 1	0.15	0.08	0.27
other method 2	0.11	0.10	0.29
ours (k=1)	0.89	0.45	<b>0.23</b>
ours (k=2)	0.09	0.23	0.70
ours (k=3)	<b>0.08</b>	0.45	0.57
ours (k=4)	0.15	0.56	0.32
ours (k=5)	0.57	0.09	0.88
ours (k=6)	0.58	<b>0.07</b>	0.89

This may seem like a simple principle to follow, but it goes wrong **a lot**. Not just in student papers, also in published research.

Here's what you might come across in a bad machine learning paper. In this (fictional) example, the authors are introducing a new method (labeled *ours*) which has a hyperparameter k. They are claiming that their model beats every baseline, because their numbers are higher (for specific hyperparameters).

These numbers create three impressions that are not actually validated by this experiment:

- That the authors have a better model than the two other methods shown.
- That if you want to run the model on dataset 1, you should use k=3
- That if you have data like dataset 1, you can then expect an error of 0.08.

None of these conclusions can be drawn from this experiment, because we have not ruled out multiple testing.

but this			
	dataset 1	dataset 2	dataset 3
other method 1	0.15	0.08	0.27
other method 2	<b>0.11</b>	0.10	0.29
k	3	5	2
ours	<b>0.11</b>	0.11	<b>0.24</b>

"The hyperparameter k was chosen based on a validation set split off from the training data. The test data was used only once."

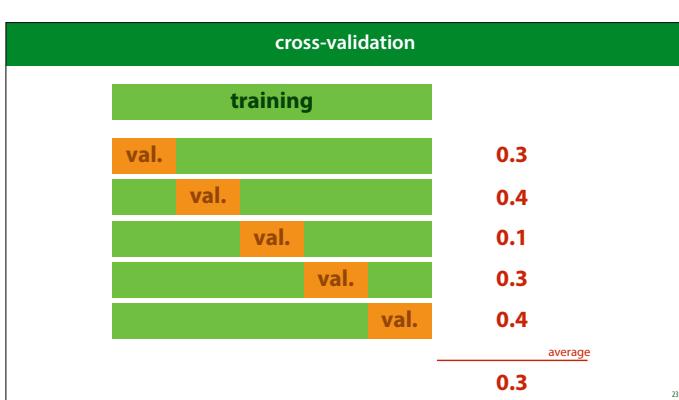
Here is what we should do instead. We should use the training data (with validation withheld) to select our hyperparameters, make a single choice for k for each different dataset, and then estimate the accuracy of only that model.

Note that the numbers have changed, because in the previous example the authors gave themselves an advantage by multiple testing. With a proper validation split, that advantage disappears. These numbers are worse, but more accurate. (I made these numbers up, but this is the sort of thing you might see)

Now, we can actually draw the conclusions that the table implies:

- On dataset 3, the new method is the best.
- If we want to use the method on dataset 3 (or similar data) we should use k=2
- If our data is similar to that of dataset 3, we could expect a performance around 0.24

Even though most people now use this approach, you should still mention exactly what you did in your report (so people don't assume you got it wrong).

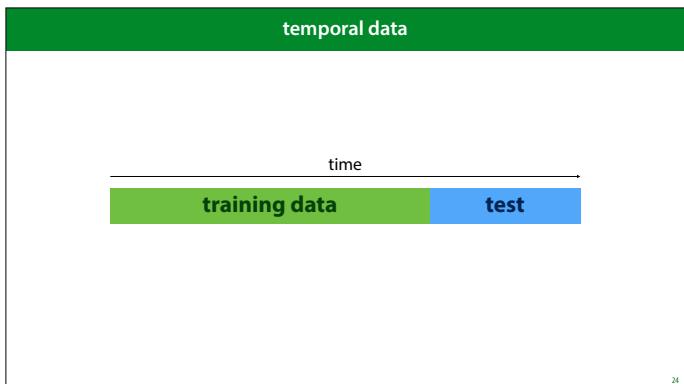


After you've split off a test and validation set, you may be left with very little **training data**. If this is the case, you can make better use of your training data by performing **cross-validation**. You split your data into 5 chunks ("folds") and for each specific choice of hyperparameters that you want to test, you do five runs: each with one of the folds as validation data. You then average the scores of these runs.

This can be costly (because you need to train five times as many classifiers), but you ensure that every instance has been used as a training example once.

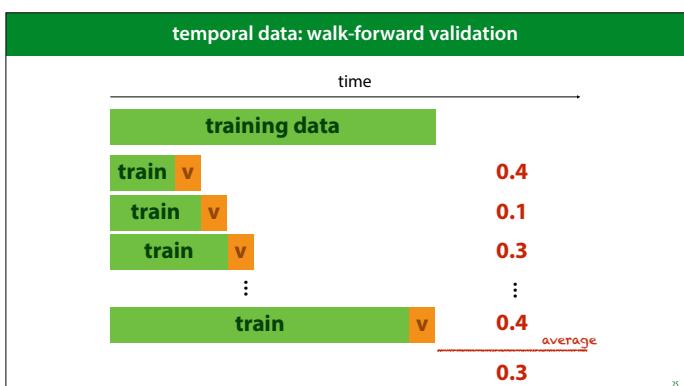
After selecting your hyperparameters with crossvalidation, you still test once on the **test data**.

You may occasionally see papers that estimate error of their finally chosen model by cross validation as well (splitting off multiple **test sets**), but this is a complicated business, and has fallen out of fashion. We won't go into in this course.



If your data has special attributes, like a meaningful temporal ordering of the instances, you need to take this into account. In the case of temporal data, training on samples that are in the future compared to the test set is unrealistic, so you can't sample your test set randomly. You need to maintain the ordering.

Sometimes data has a timestamp, but there's no meaningful information in the ordering (like in email classification, seeing emails from the future doesn't usually give you much of an unfair advantage in the task). In such cases, you can just sample the test set randomly.



If you want to do cross-validation in such time sensitive data, you'll have to slice the dataset like this.

Evaluation is a simulation of production.  
Validation is a simulation of evaluation.

26

In general, don't just apply split testing and cross validation blindly. *Think* about how you will ultimately train and use your model "in production". Production may be an actual software production environment, or some other place where you intend to employ your model. Your evaluation on the test set is essentially a *simulation* of that setting.

If you're doing something in evaluation that you won't be able to do in production (like training on instances from the future), then you are cheating your evaluation.

Your **validation** is essentially a simulation of the **evaluation**. If you want validation results that accurately predict your evaluation results, then your validation should mimic the evaluation as closely as possible.

Here, however, you are allowed to deviate a little. For instance, you can make your validation data a little smaller than your test data. This is a tradeoff: you are reducing the certainty of your validation results, but you are gaining a little extra training data, which will improve your results in the end. Such tradeoffs are fine, so long as you are honest in

your final evaluation on the [test data](#).

In general, when in doubt **make sure that the evaluation setting accurately simulates production, and that the validation setting accurately simulates the evaluation setting.**

## which hyperparameters to try?

Up to you:

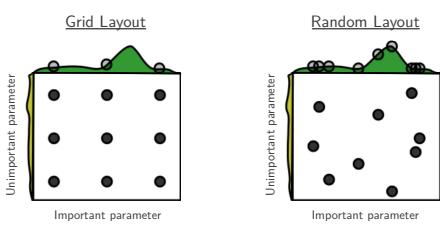
- trial-and-error (intuition)  
probably the most common approach
- grid search  
define a finite set of values per hyperparameter and try all combinations.
- random search ([remember?](#))

So, now that we know how to experiment, what experiments should we run? Which values should we try for the hyperparameters? So long as we make sure not to look at our test set, we can do what we like. We can try a few values, we can search a grid of values exhaustively, or we can even use methods like random search, or simulated annealing.

*We should only be cautious not to try too many different hyperparameter values if our test and validation sets are small.*

It's important to mention: **trial and error is fine, and it's the approach that is most often used.** It's usually the most effective, because you (hopefully) have an intuitive understanding of what your hyperparameters mean. You can use this understanding to guide your search in a way that automated methods can't.

## random samples vs. grid search



source: [Random search for hyper-parameter optimization](#), Bergstra and Bengio JMLR 2012

28

If you are going to use some kind of automated search, trying a bunch of different combinations of hyperparameter values, and then trying random samples in the hyperparameter space is often better than exhaustively checking a grid. This picture neatly illustrates why. If one parameter turns out not to be important, and another does, a grid search restricts us to only three samples over the important parameter, at the cost training nine different models.

If we randomize the samples, we get nine different values of each parameter at the same cost.

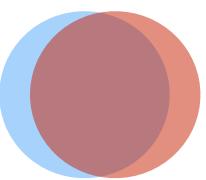
source: <http://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a.pdf> (recommended reading)

**Model Evaluation**  
Part 2: Statistical Testing

Machine Learning  
mlvu.github.io  
Vrije Universiteit Amsterdam

|section|Statistical Testing|  
|video|https://surfdrive.surf.nl/files/index.php/s/  
AoCsdmxAmof17Y9/download|

### Machine Learning vs. Statistics



**Statistics**      **ML**

**Stats but not ML:** Analyzing research results. Experiment design. Courtroom evidence.  
**More ML than Stats:** Spam classification, movie recommendation.,

30

As noted in the first lecture, statistics and ML are very closely related. It's surprising then, that when we perform ML experiments, we use relatively little of the statistics toolkit. We don't often do **significance tests**, for instance.

### should we do statistical tests at all?

- Makes ML experimentation difficult. Lots of disagreement.
- People overestimate the value of statistical analyses.
- Does not promote the best methods
- The ultimate validation of research is REPLICATION

On the appropriateness of statistical tests in machine learning, Janez Demšar, 2008  
 Machine Learning as an Experimental Science (Revisited), Chris Drummond, 2006

31

Should we be doing more statistics on our own experiments?

There is a lot of disagreement. Hypothesis testing comes with a lot of downsides. Given that we usually have very big sample sizes (10 000 instances in the test set), our efforts may be better spent elsewhere.

Another consideration is that the ultimate validation of research is *replication*, not statistical significance.

Somebody else should repeat your research and get the same results. Because all of our experimentation is computer code, a basic replication could be as simple as downloading and running a docker image. After that it's easy to try the same on new data, or check the model for bugs.

*In practice, replication can be a real nightmare, even in our field.*

Since the community is so divided on the question, we won't emphasize statistical testing too much in this course.

However, there are a few important statistical concepts to be aware of, even if we don't use the whole statistical toolbox to interrogate them rigorously.

true metric vs. estimate	
unobservable	observable
data distribution: $p(x, t)$ one instance $x$ and its class $t$	- sample → dataset
true accuracy of $C$ probability that $C(x) = t$ under $p$	sample accuracy of $C$ proportion of test set that $C$ classifies correctly
e.g. the mean height of a Dutch person	→ the mean height of 100 randomly sampled Dutch people

The first is the difference between the **true metric** of a problem or task, and the *value you measure*. This is a very basic principle in statistics. For instance, we can't observe the mean height of all Dutch people currently living, but we can take a random sample of 100 Dutch people, and use their average height as an **estimate of the true average height**.

To translate this to machine learning, let's take classification accuracy as an example.

We usually imagine that the data is **samples** from some distribution  $p(x)$ . In this view, we're not really interested in training a classifier that does well on the dataset we have, even on the test data. What we really want is a classifier that does well on any data sampled from  $p(x)$ .

Imagine sampling one instance from the data distribution and classifying it with some classifier  $C$ . If you do this, there is a certain probability that  $C$  will be correct. This is called the **true accuracy**. It is not something we can ever know or compute (except in very specific cases). The only thing we can do is take a large number of samples from  $p(x)$ , classify them with  $C$ , and approximate the true accuracy with the relative frequency of correct classifications in our sample. This is what we are doing when we compute the accuracy of a classifier on the test set or the validation set: **we are estimating the true accuracy**. To explicitly distinguish this estimate from the true accuracy, we sometimes call this the **sample accuracy**.

The accuracy is just the simplest example. We can apply the same idea to any metric, like the MSE loss of a regression model, or the many metrics for classifiers we will see in the following videos. They all have a **true value** defined on the data distribution, which we can't observe, and an estimate which we can compute from the test set.

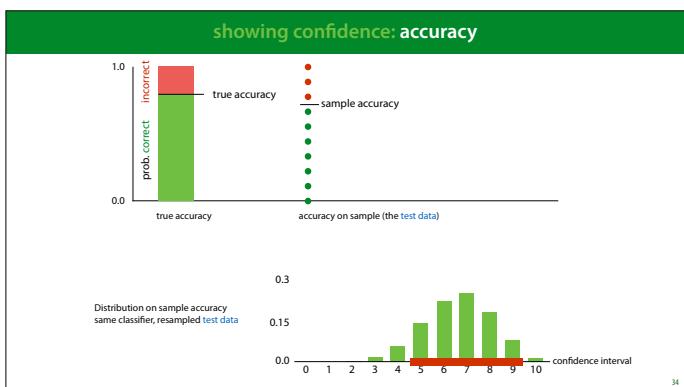
Can the observed results be attributed to *real characteristics* of the models under scrutiny or are they observed *by chance*?

33

This brings us to the main question that statistical analysis is meant to answer. If we do things properly, and we have a large dataset, our estimate will be close to the true value. Often, we can even prove how close it is likely to be. But there will be some difference, which will be entirely random.

So, when we estimate the test accuracy of models A and B and we see that classifier A is better than classifier B because their estimated accuracies on the test set are .997 and .998 respectively, can we really trust that statement? Maybe this random noise we get when we compute the estimate of the accuracy caused this difference. In other words, **how sure can we be, from these values, that the true accuracy of A is also higher than the true accuracy of B?**

quote source: [http://www.icmla-conference.org/icmla11/PE\\_Tutorial.pdf](http://www.icmla-conference.org/icmla11/PE_Tutorial.pdf)



One way of doing this is to compute a **confidence interval**. Here we see the process of computing a sample accuracy in a simple animation: we start with the true accuracy (for some given classifier, on the data distribution) which is somewhere between 0 and 1. We sample a bunch of points from the data distribution (our test set), and take the relative frequency of correctly classified instances as the **sample accuracy**.

Here, in the top half of the slide we model the process of taking one instance of our test set and seeing whether the classifier classifies it correctly as a single random draw resulting in the outcome **correct** or **incorrect**. *We'll see in the next lecture that this type of distribution is called a Bernoulli distribution.*

The whole process of sampling the entire test set and computing the sample accuracy is also a random process. If we were to repeat it, sampling a new test set, we'd get a different value for the sample accuracy. To simplify this, we can look at the total number of instances in our sample that the classifier classified correctly (so we don't divide by N). In that case, it turns out we can work out the distribution of this process as well: the number of "correct"s we get in N samples from a Bernoulli distribution forms what is known as a Binomial distribution.

The technical details aren't important. The main message is that we can define precisely what distribution we can expect on the value of the sample accuracy if we keep the classifier and the true accuracy fixed, but resample the test data. In this case, we see that for a true accuracy of 0.8, and a test set of N=10 instances, we are most likely to see that the classifier correctly classifies 7 instances in our sample (as we did in the top half of the slide). However, it's also perfectly possible to see 5, 6, or 8 instances classified correctly.

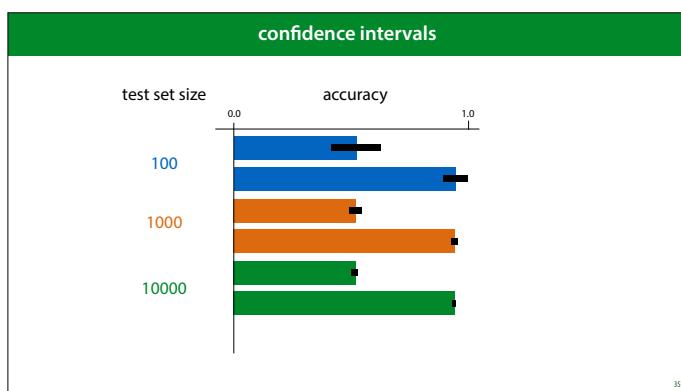
*This is a complicated picture with multiple random processes going on. Take a little time to wrap your head around this before moving on.*

Imagine that if we have two classifiers, and we compute sample accuracies for both on a test set of 10 instances. We get 5/10 correct for one classifier and 8/10 correct for the

other. Can we conclude that the first is definitely worse than the second? What we see here is that if they *both* have a true accuracy of 0.8, it would be perfectly likely to see these numbers. In short, with such little test data, we have much uncertainty around our estimate of the true accuracy.

One way to quantify this uncertainty is to draw a **confidence interval**. This is simply any interval on the values of our sample accuracy that captures a given proportion of the probability mass, usually 95%. You can draw the confidence interval anywhere you like so long as you decide how to draw it before seeing the data. The most common approach is symmetrically around the mean, but you can also start at the mean and extend it to the right as far as possible or to the left as far as possible.

The confidence interval captures our uncertainty. We don't know the true value of the accuracy, and we can't estimate it very accurately, but we know it's most likely one of these values.

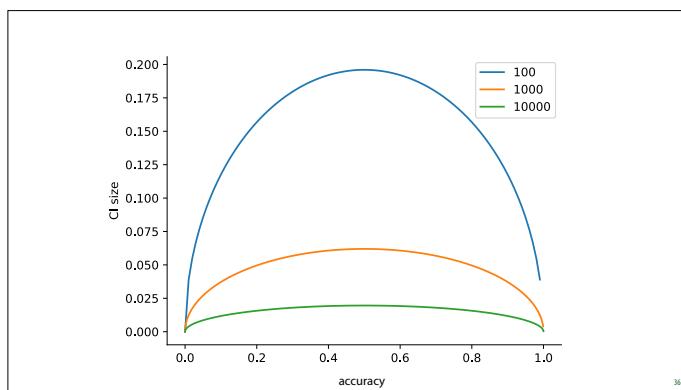


The size of this confidence interval depends on two factors: the true accuracy and the size of the test set. Here are some examples for different accuracies and test set sizes.

This tells us that if the true success probability (accuracy) of a classifier is 0.5, and the test set contains 100 examples, our confidence interval has size 0.2. This means that even if we report 0.5 as the accuracy, we may well be wrong by as much as 0.1 either side.

Even if these confidence intervals are usually not reported, you can easily work them out (or look them up) yourself. So, if you see someone say that classifier A is better than classifier B because A scored 60% accuracy and B score 59%, on a test set of 100 instances, you have reason to be sceptical.

In short, this is why large test sets are important. Without at least 1000 instances in your test data, you cannot reliably tell the difference between two classifiers.



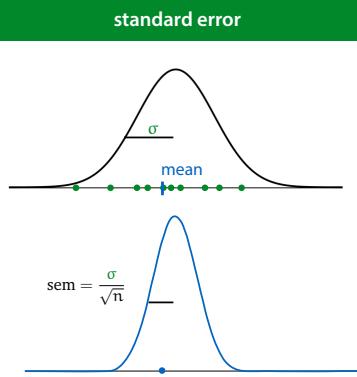
Here are the full curves, in case you ever need to look it up.

Confidence depends on *the size of the test set*.

Avoid small test sets.

If you can't, look into Alpaydin's 5x2 F test  
<https://www.cmpe.boun.edu.tr/~ethem/files/papers/NC110804.PDF>

If you don't have the luxury of a large test set, you may need to do some statistical testing to see whether the effect you've observed (classifier A is better than classifier B) is genuine or down to random chance. It's generally accepted that Alpaydin's 5x2 cross validation is the best test for this purpose. It's out of scope for this course, but follow [the link](#) if you run into this problem.



If we are computing a continuous value, like the mean squared error loss of a regression model, the same principle applies. For now, let's forget about the details and assume that we are computing some number representing the difference between the true regression value of an instance and the predicted regression value. We'll call this number **m**.

*In this case m is the error of a model, but it could be any measurement of any phenomenon.*

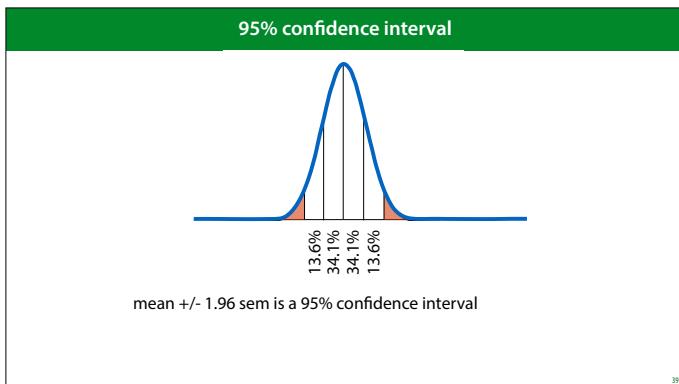
We can often assume that m, computed for a single randomly sampled instance, is normally distributed. The likely values are clustered around a mean value, or expectation of m. This is the distribution drawn in the top half of the slide.

If we sample a test set and compute the mean of m for all instances in the test data, we get an estimate for the true expectation of m. This is an estimate of the mean of the distribution at the top. Note that the estimated mean is slightly different from the true mean.

This, again, is a random process. If we sample another test set, keeping the model and the true mean fixed, we get a slightly different estimate for the mean. The distribution on the values we get for the **sample mean** is drawn at the bottom. Note that its mean is the same as the true mean, but its peak is more narrow.

*This is not a normal distribution, but a so called Student's t distribution. For test sets larger than ~30 instances, the difference becomes negligible.*

The standard deviation of the distribution at the bottom is the variance of the one at the top, divided by the square of the number of instances in our test set. The more instances, the narrower the peak becomes, and the less uncertainty we have around our estimate. This standard deviation is called the **standard error of the mean (sem)**.



As you may know, the region of four standard deviations around the mean of a normal distribution contains roughly 95% of the probability mass. This means that the interval from two standard errors to the left of your mean to two standard errors to the right of your mean is a 95% confidence interval.

confidence interval <i>estimates</i>	
unobservable	observable
data distribution: $p(x, t)$ Instances $x$ and classes $t$	dataset
true accuracy of $C$ probability that $C(x) = t$ under $p$	sample accuracy of $C$ proportion of test set that $C$ classifies correctly
<i>true</i> confidence interval	<i>estimated</i> confidence interval

40

These are good concepts to help us think about what we are measuring when we compute metrics on our test data. But it's important to realize that these are unobservable values. We can only work out the true confidence interval for the sample accuracy if we know the true accuracy. This puts the confidence intervals we've talked about so far in the unobservable column.

So what about all those confidence intervals you see reported in the literature? These are estimated confidence intervals. They are usually computed in the same way as the true confidence interval, but wherever we need the true value of some metric, we replace it by its estimate. This gives us a confidence interval that isn't as correct as the true confidence interval, and one that would change slightly if we were to resample the test set, but we can at least compute it, and it generally behaves in roughly the same way as the true confidence interval.

about confidence interval estimates
Don't say: <b>the probability that the true mean is in <i>this</i> confidence interval is 95%.</b>
Do say: <b>If we repeat the experiment many times, computing the confidence interval each time: the true mean would be inside the interval in 95% of those experiments.</b>
<small>The confidence interval changes from experiment to experiment, not the true mean.</small>
The estimated confidence interval for the mean is a statistic on the data, just like the mean itself or the standard deviation.

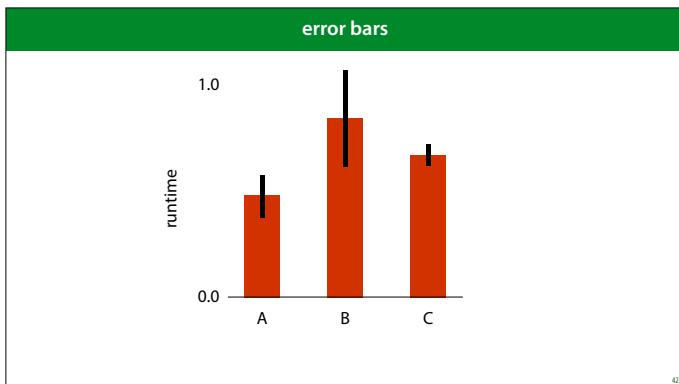
41

When we use the phrase *confidence interval* to refer to this kind of estimate, it's important to speak about it correctly.

There is no probability associated with the true mean at all. It is simply an objective, determined value (which we don't know). The probability comes from sampling, and from computing the interval from a sample.

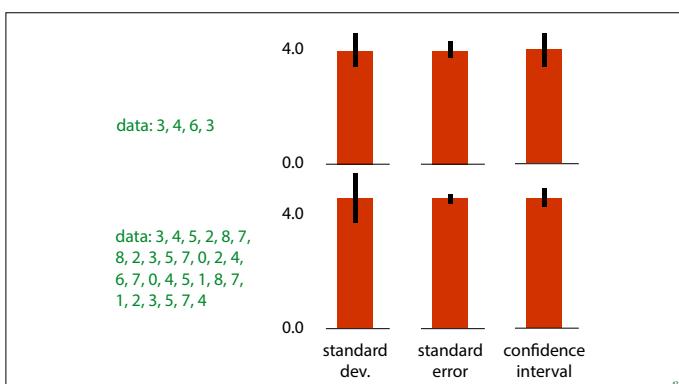
So instead of having a fixed interval, with the true mean jumping around probabilistically, we have a fixed true mean around which we get an interval that jumps around if we resample the data. The probability of it jumping so much that it no longer contains the true mean is 5%.

*This is typical frequentist agonizing over what the phrase "probability" means. Confidence intervals are a uniquely frequentist tool. If the distinction on the slide doesn't make sense to you, look back to the probability video in the preliminaries.*



A very relevant question is how do you interpret **error bars**?

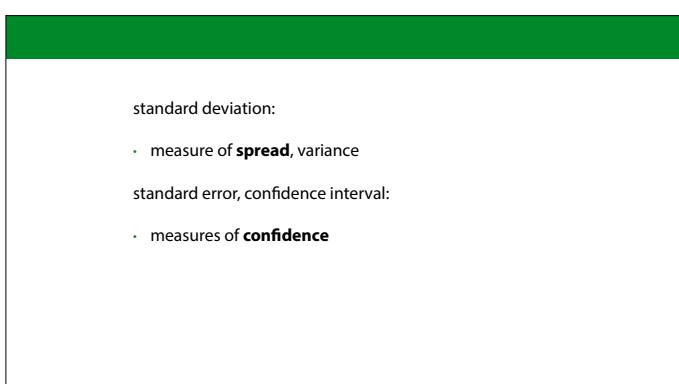
If you see a picture like this, showing the mean runtime of an experiment, measured for three models, and averaged over a number of runs, what would you imagine the error bars denote? We've seen standard deviations, standard errors and confidence intervals. What do error bars represent?

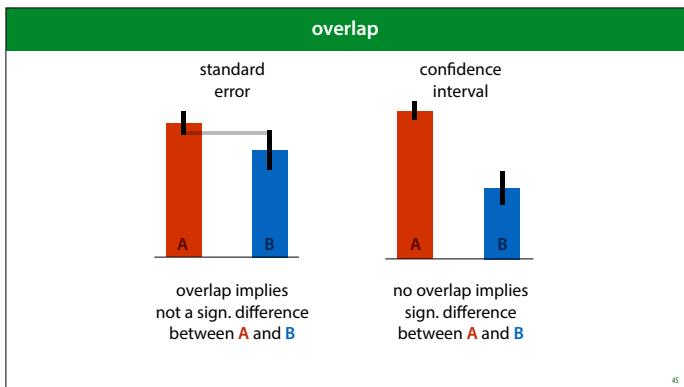


The truth is that there is no standard definition for what error bars denote, and if the authors didn't specify what their error bars indicate, the authors messed up.

These are the three most common options. If you didn't quite get all the details of the previous slides, this slide illustrates the most important distinction: If we sample more data, the estimate of our **standard deviation** becomes *more accurate*. It's an estimate of a property of our data distribution. The error bars representing standard deviation don't get smaller, they just get closer to their correct size (which may be very wide around the head of the bar).

The standard error and the confidence interval are indicators of how confident we are about our estimate of *the mean of our data*. For these, the more data we have, *the smaller they get*. As we saw earlier, *under the right assumptions*, the 95% confidence interval is roughly twice the standard error.





Under certain assumptions, the standard error of two estimates can tell you whether a Student's t test allows you to reject the null hypothesis **that the two distributions are the same**. This would allow you to say, for instance, that there is a statistically significant difference between the accuracy estimates on two classifiers. As we said before, such testing is not very popular in machine learning, so we won't go into it in detail, but this can offer an additional perspective on error bars.

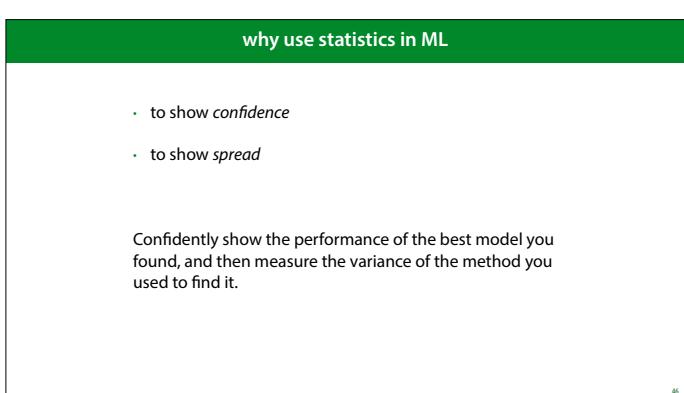
*We won't discuss the Student's t test in detail, but it's a common test for comparing two normally distributed values.*

Say you plot the mean squared error for regression models **A** and **B**, together with some error bars. Does the fact that the error bars overlap or not tell you whether the measured difference between the two models is statistically significant? That is, does it indicate whether we can conclude that **A** is a better model than **B**?

Yes, but we have to be careful. For standard error bars, the existence of overlap implies that there is no significant difference between the two effects. That is, the possibility that the difference is due to random chance is high, and a repeat of the experiment on new data may show a different result. If you plot confidence interval error bars, and there is *no overlap*, you may conclude that the difference between the models is significant. If you repeat the experiment on fresh data, it is very likely that model **A** would beat model **B** again.

In both cases, the converse does not hold. If the SEM error bars do not overlap, there may or may not be a significant difference. If the confidence interval error bars do overlap, there may still be a significant difference, depending on how much they overlap.

*All of this requires the assumption that the original values for which the bar indicates the mean are normally distributed.*



All of this was about **showing confidence**: showing how *reliable* our numbers are as estimates of the true values that we can't observe.

**Showing spread** is more about providing insight to the reader. Say I train a classifier by gradient descent. If I have a big **test set**, I can very *confidently* measure and report the accuracy of this particular classifier. However, gradient descent uses *random* initialization. If I repeat the training process, I may end up in a different local minimum, and get a different classification performance. It's likely that I also want to communicate how much the measured performance is dependent on *this* randomness. In short, how lucky did we get in our choice of classifier?

## showing spread

Sources of randomness:

- Data sampling
  - Search algorithm (i.e. initializing gradient descent)
- Report standard deviation, describe *what you repeat*.
- How do you repeat data sampling?

If we have a large enough test set, we know that the confidence interval for our measurement of the performance is small enough. But we also want to know how much the randomness in our process affects the result. What is the probability that repeating the process (on the same data, or on new data) produces wildly different results?

For factors like the initialisation of gradient descent, this is easy to test: you just rerun a few times on the same data. But how do you test how robust the results are against **sampling a new dataset?**

## resampling

Cross validation again, on the whole data set.

Stratified cross-validation (keeps the class proportions the same in all folds).

Leave-one-out cross-validation, a.k.a. the jackknife method.

Slight bias: smaller datasets.

The cross validation that we have already seen provides some indication of spread. If we do five-fold cross validation, we are repeating our training five times on slightly different datasets. This provides some indication of how differently the algorithm may perform if we repeat on new data.

Note, however that it's not quite a perfect simulation of resampling our data: the datasets are slightly smaller, and there is a lot of overlap between the five different datasets. If we want more repeats, we get more overlap and a smaller test set, making this bias hard to control.

## bootstrapping

Sample, with replacement, a dataset of the same size as the whole dataset.

On average, about 63.2% of the dataset will be included. The rest will be duplicated instances.

Each bootstrapped sample lets you repeat your experiment.

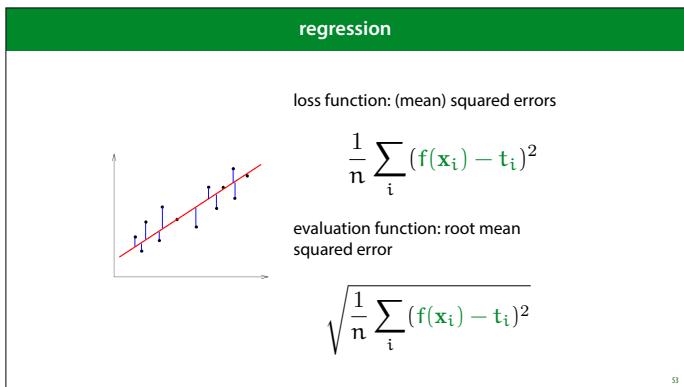
Note that some classifiers will respond poorly to presence of duplicate instances.

Better than cross validation for small datasets.

A different approach is bootstrapping. Here, we resample the data **with replacement**. This allows us to sample a dataset that is exactly the same size as the original data. We can also resample as often as we'd like, without affecting the size of the test data or the amount of overlap between the datasets.

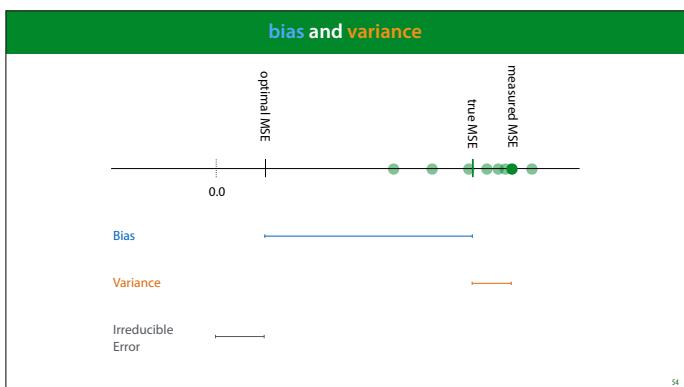
We will see in a later lecture that bootstrapping approximates the data distribution in a very precisely defined way.





One thing to pay attention to is that if you use MSE loss, you may want to *report* the square root (the RMSE). The RMSE is minimised at the same places as the MSE, but it's easier to interpret, because it has the same units as the original output value.

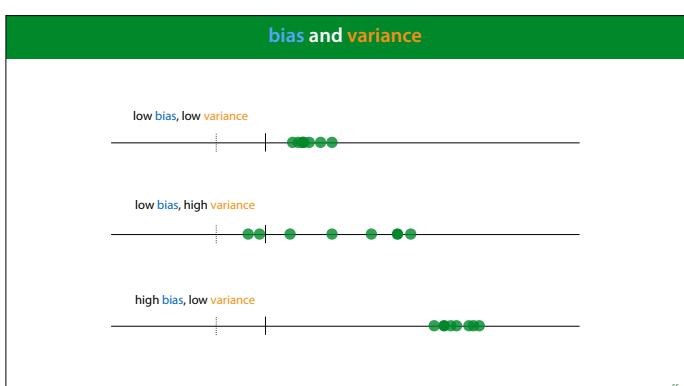
For instance, if your outputs are in meters, then your MSE is measured in square meters, but your RMSE is also measured in meters.

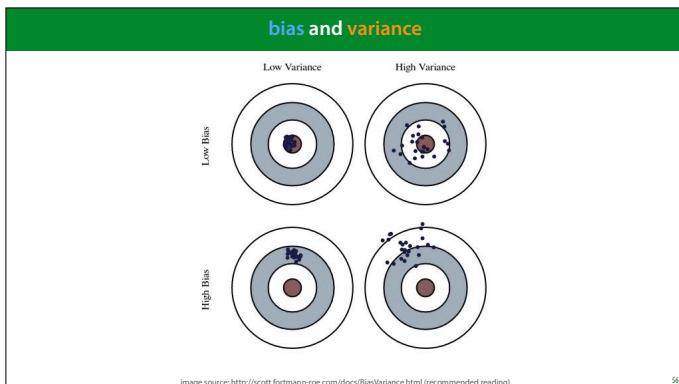


Bias and variance are good concepts to keep in mind when thinking about regression performance. The bias is the difference between your optimal MSE and the true MSE. This is the part of your error that is down to fundamental problems with your model: for instance, you are fitting a line through data with a parabolic pattern. This part stays the same if you resample your data.

The variance is the difference between the true MSE and the measured MSE. This is the part of your error that is down to aspects of the random sampling of the data. This part changes when you resample your data.

Normally, we train a regression model once, and get one MSE value. This gives us one dot on the axis above. Without repeating the process on freshly sampled data, we can't tell how our error falls apart in bias and variance. However, we can usually get some contextual clues, or investigate using resampling.

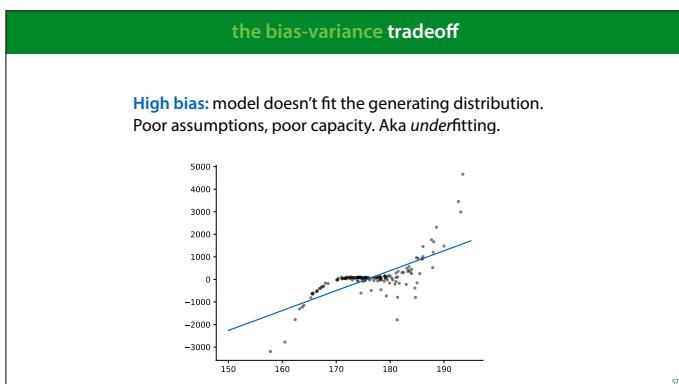




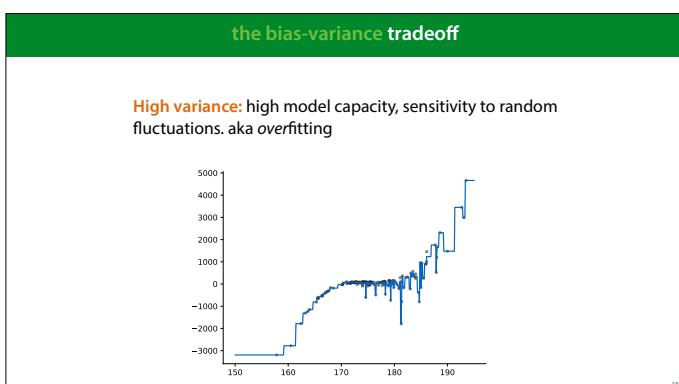
Here is a metaphor that is often used to describe bias and variance: a dartboard.

Remember, this is a metaphor for our RMSE error estimate. That means that normally, we have only one dart and we can't tell whether our error is due to high bias or high variance.

image source: <http://scott.fortmann-roe.com/docs/BiasVariance.html> (recommended reading)



High bias tends to happen when the model is too simple to follow the true "shape" of the data. Linear models in low-dimensional spaces often have this problem. Here, we see that the data has a slight curve, which is clearly part of its natural pattern, and something the model should learn. Since it's restricted to a line, however, it cannot make this shape.



High variance happens when the model has the capacity to follow the shape of the data perfectly, but it does it so perfectly that it tends to get thrown off by small fluctuations.

Here, the model doesn't just follow the natural curve of the data, it goes in and out of every random fluctuation to model every single point perfectly.

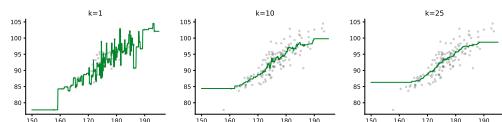
Even though this model (a regression tree) fits the data perfectly, if we resample the data, we are stuck with all sorts of weird peaks that won't fit the new data. This is where the *variance* comes from. The true error varies wildly, because the model captures every single random fluctuation in the training data. These fluctuation will cause a large error in the test data, which we put down to variance rather than bias.

## making the tradeoff

Reducing **bias**: increase model capacity, increase features.

Reducing **variance**: reduce model capacity, add regularization, reduce tree depth.

k-NN regression: increase k to increase **bias**, decrease **variance**.



We will see techniques for all of these in the coming weeks. Note that often, it really is a tradeoff: reducing the bias, increases the variance and vice versa.

For some algorithms, there is a single parameter that allows us to make the bias/variance tradeoff. kNN is one example: low k values give us high variance, high k values give us high bias.

*In the strictest definition of bias and variance, these concepts only apply to the mean squared error, where they explicitly appear as terms if you rewrite the error in a certain way. However, in general machine learning parlance, the terms are applied to any regression error, and they are roughly synonymous with under- and overfitting.*

## lecture 10: ensembling

Combining models for **variance** reduction and for **bias** reduction.



In a later lecture, we'll look at **ensembling**. This is a method that allows us to combine different models, so as to control the problems of high bias and high variance.

image source: <https://www.toptal.com/machine-learning/ensemble-methods-machine-learning>

image source: <https://www.toptal.com/machine-learning/ensemble-methods-machine-learning>

60

## evaluating classification

Class imbalance, cost imbalance

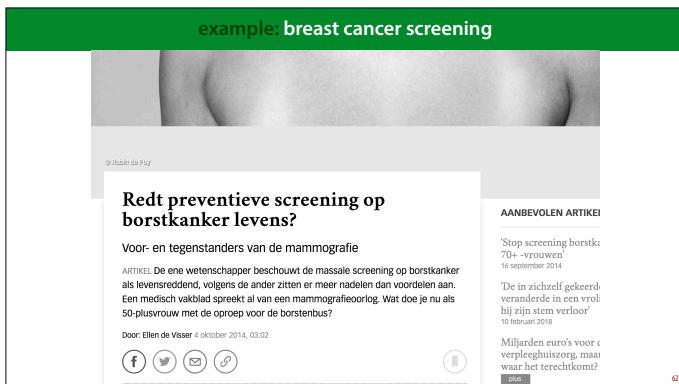
Confusion matrix

True positive rate, True negative rate  
More in the next video

Precision, recall

Let's now move to classification. We'll start by explaining these four topics.

61



Here's one example domain: breast cancer screening

A recurring discussion in the Dutch media is the question whether all women over 50 be screened for breast cancer. This is an analogy for classification: the instances are people and the target label is “**has cancer**” or “**has no cancer**.” You may think that this is a no-brainer: the more tests we do, the more lives saved. But we need to take into account how good the classifier is, and how bad the consequences are of it making a mistake.

The first problem this example illustrates is **class imbalance**. Unlike the classifiers we've seen so far, this example has, thankfully, far more **negatives** than **positives**. In a given year the people in this age group for whom breast cancer will manifest is about half a percent. This includes men, for whom breast cancer is rare (but not impossible), so when we talk about screening for women over 50, we should probably assume about 1% positives.

This means that we need to be very careful when interpreting the reported performance of any breast cancer detection method. We may think that an accuracy of 99% sounds impressive, but this is the accuracy we would get if we just called everything negative: that is, if we just diagnosed all women with no cancer.

source: <https://www.volkskrant.nl/wetenschap/redt-preventieve-screening-op-borstkanker-levens~a3761451/>

## Is an error of 0.01 good?

Imagine that somebody tells you about a machine learning project they're doing, and they proudly state that they get a classification error (on their validation set) of **0.01** (1% of the validation set is misclassified). Should you be impressed?

The answer is it depends. The first thing it depends on is the **class imbalance** in the data.

This AI judge correctly predicts court case results 80% of time

A statue of blind justice outside the Albert V. Bryan United States Courthouse in Alexandria, Virginia. Image Credit: Tim Evanson

A team of computer scientists and legal professionals has created artificial

So the next time you see a headline like this, your first question should be: what was the class distribution in the training data? If 90% of the cases in the training data are acquittals, this is not a very impressive result.

As it happens, in this case the classes were balanced 50/50, so 80 percent is at least notable. However, now we have a classifier trained on artificially balanced data. In a production environment (whatever that means here), the classes are likely not balanced 50/50, so this specific classifier will be of no further use.

*Here is the original paper: <https://peerj.com/articles/cs-93/#fn-6> There are some issues with this research beyond the class balance.*

### is 1% error good? it depends

**Class imbalance** How much more likely is a **Positive** example than a **Negative** example?

**Cost imbalance** How much **worse** is a mislabeled **Positive** example than a mislabeled **Negative** example?

Another reason to mistrust accuracy is **cost imbalance**. In breast cancer screening are two types of misclassification: diagnosing a healthy person with cancer and diagnosing a person with cancer as healthy. Both come with a cost but not the same cost.

We can either miss a cancer diagnosis (*a false negative*), which means the cancer will be caught much later and be much harder to treat. However, diagnosing a healthy person with cancer (*a false positive*) means they will be sent for unnecessary invasive testing and suffer great psychological stress. The cost of this is much less than the cost of missing a positive, but it isn't zero.

This is what the discussion in the media centers on. If the screening causes many more false negatives than false positives, it may mean that the cost to human lives balances out, and the net effect of the screening isn't to save more lives.

There is of course, also the financial cost of screening a large part of the population. Most people would put any financial cost far below the cost of a human life, but if it turns out that the classifier is weak enough that the gains in human lives saved are negligible, we'd look at the financial cost and say the program would need to end.

## cost imbalance

**disease diagnosis** Sending a sick person home vs applying invasive tests to a healthy person

**spam classification** Deleting a valid email vs showing the user a single spam email

**detecting financial fraud** Having an expert review a non-fraudulent transaction vs missing a fraud in progress

Domain-specific evaluation function: dollars lost, time lost, lives lost, etc.

Here are some examples of problems with cost imbalance

In all these cases, one misclassification one way costs much more than a misclassification the other way. But both cost *something*. The time of an expert reviewer is not free, even though five minutes of his time may be much cheaper than the cost of letting a single fraud go unchecked. In such a case, you may decide that missing one fraud is as costly as having an expert review 500 harmless transactions. This is then the general balance you are hoping for: one false negative for every 500 false positives.

If you're lucky, both types of misclassification have the same unit, and you can turn your error (an estimate of the number of misclassifications) into a domain specific evaluation function (like estimated dollars lost, or time saved). **You simply assign a cost to each type of misclassification, and multiply it by how often that misclassification occurs in the test set.** The total is the evaluation function you want to minimize.

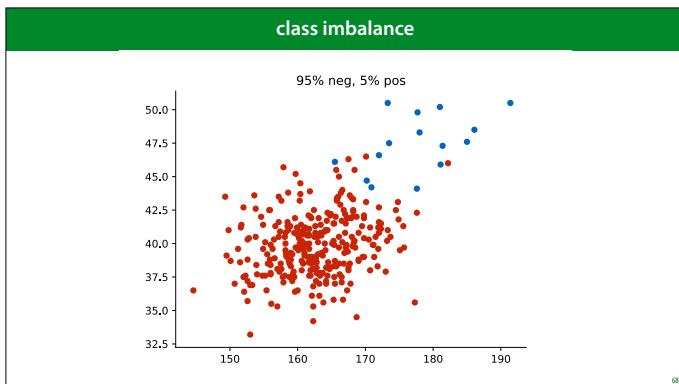
If the units are not the same (money saved vs. lives saved) making such a choice can seem very unethical if you're literally equating a human life with an amount of money. On the other hand, any classifier you decide to deploy will *implicitly* make such a choice even if you don't do the sums yourself. Even if you decide not to use machine learning, the alternative (a doctor using their own judgement) is also a "classifier", with its own cost balance.

## social impact

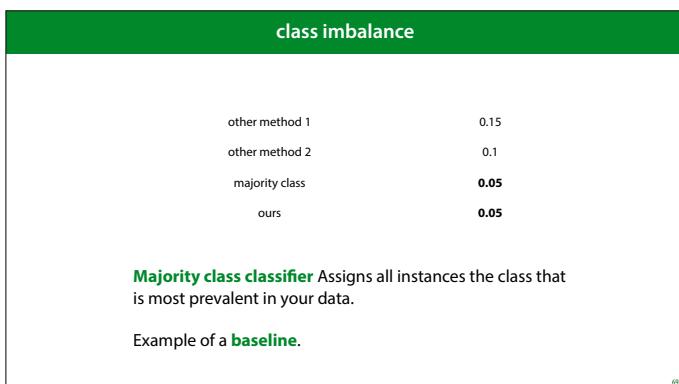


Cost imbalance is particularly important when we consider matters of social impact. If we predict a person's sex from their physical appearance perfectly, and we use that as a prediction for their gender, we may easily achieve 99% accuracy.

However the 1% we then misclassify is precisely that part of the population for which gender is likely to be a sensitive attribute. Just because our classifier has high accuracy, doesn't mean it can do no harm. In a large part because the mistakes it makes are not uniformly distributed. They are focused squarely on the vulnerable part of the population.

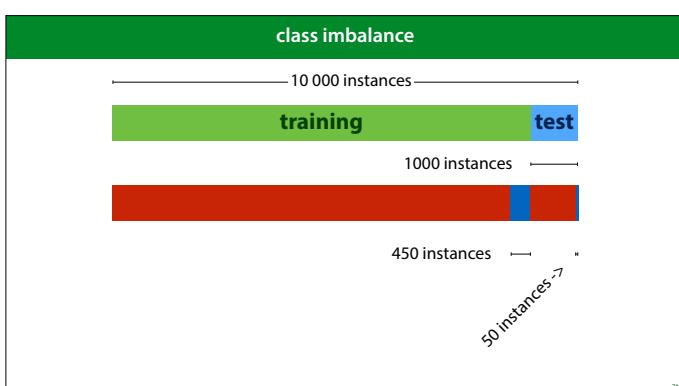


Here is a pretty imbalanced dataset (though still not as imbalanced as the cancer/not cancer problem). It looks pretty difficult. What would be a good performance on this task?



As we've seen, even though an error of 0.05 might sound pretty good, but on an imbalanced dataset like this, there is a very simple classifier that gets that performance easily. The classifier that assigns anything the class with the most instances. We call this the **majority class classifier**.

The majority class classifier is an example of a **baseline**, a simple method that is not meant to be used as a real model, but that can help you calibrate the performance scores. In this case, it tells you that you're really only interested in the error range from 0 to 0.05. Any higher error than that is pretty useless.



Here is another way that class imbalance can screw things up for you. You might think you have a pretty decent amount of data with 10 000 instances. However if you split off a test set of 1 000 instances, you'd be left with just 50 instances of the **positive** class in your data. Practically, your final evaluation will just be a question of how many of these 50 **positives** you detect. This means that you can really only have 50 "levels of performance" that you can distinguish between.

You can make a bigger test set of course (and you probably should) but that leads to problems in your training data. Since you're essentially building a detector for **positives**, it doesn't help if you can only give it 100 examples of what a **positive** looks like.

In the next lecture, we'll look at some tricks we can use to boost performance on such imbalanced data.

## other performance metrics

Confusion matrix

Precision, recall

True positive rate, false positive rate

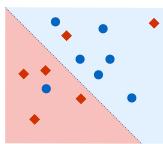
ROC plot, Coverage matrix, Area under the curve

[Next video](#)

The best thing to do under class and cost imbalance, is to look at your performance in more detail. We'll look at six different ways to measure classifier performance.

Most of these are only relevant if you have class or cost imbalance. If you have a nice, balanced dataset, it's likely that error or accuracy is all you need.

## confusion matrix



		predicted		
		pos	neg	
actual	pos	6	1	7
	neg	2	5	7
		8	6	

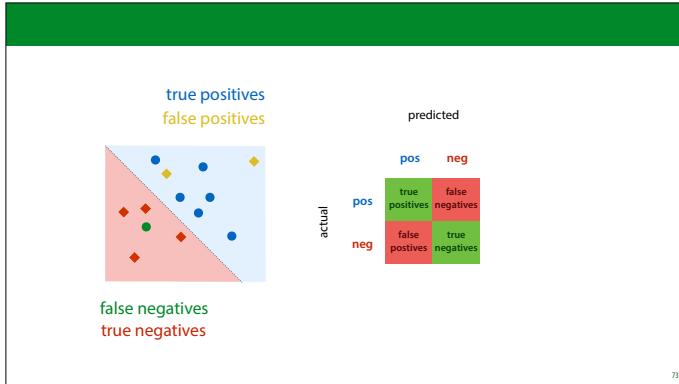
This is a **confusion matrix** (also known as a contingency table). It's simply a table with the actual classes on the rows, and the predicted classes on the columns, and a tally in each cell of how often each actual class is given a particular prediction. On the diagonal we tally all the correct classifications and off the diagonal we tally all the possible mistakes.

A confusion matrix doesn't give you a single number, so it's more difficult to compare two classifiers by their confusion matrices, but it's a good way to get insight into what your classifier is actually doing.

Note that for a binary classification problem, we are getting the two types of mistakes (false positives and false negatives) along the second diagonal. If we have cost imbalance, the balance between these two values gives us a quick insight into how well the classifier is aligned with our estimate for the misclassification costs.

You can plot the confusion matrix for either the [training](#), [validation](#) or [test](#) data. All three can be informative.

The margins of the table give us four totals: the actual number of each class present in the data, and the number of each class predicted by the classifier.

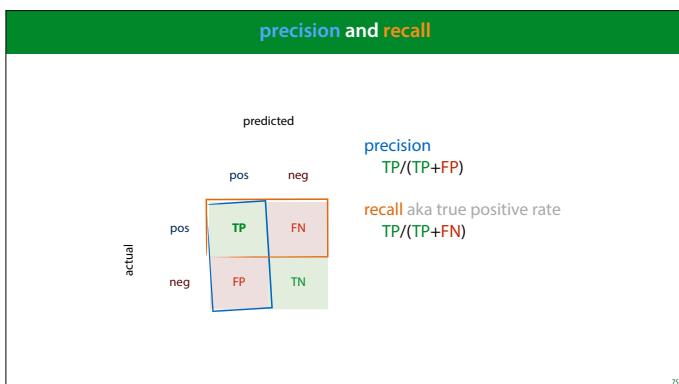


We call accurately classified instances **true positives** and **true negatives**. Misclassifications are called **false positives** and **false negatives**.

with class imbalance			
		predicted	
actual	pos	pos	neg
		385	0
	neg	15	0
		385	15
		400	0

74

Here we see the confusion matrix for the majority class baseline (the classifier that calls everything positive) in a problem with high class imbalance.

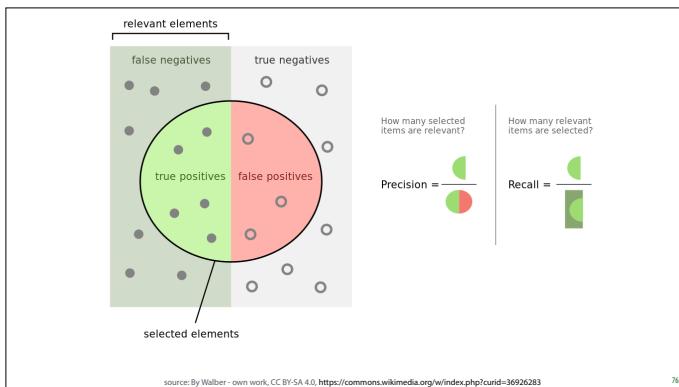


Precision and recall are two metrics that express a tradeoff between the two types of mistakes.

**Precision:** what proportion of the returned positives are actually positive?

**Recall:** what proportion of the existing positives did we find?

The idea is that we usually want to find as many positives as possible, so we should be eager to label things positive, increasing the recall, but if we are too eager, we will label lots of negatives as positive as well, which will hurt our precision. Our main challenge in designing a classifier in the face of cost and class imbalance, is to find the right tradeoff between precision and recall.



It always takes me a minute to figure out what precision and recall mean in any given situation, and I usually consult this diagram from Wikipedia to help me out.

The idea is that the goal of the classifier is to select the positives in the dataset. The more it selects, the higher its recall, but the lower its precision, as more negatives end up in the selection.

source: By Walber - own work, CC BY-SA 4.0, <https://commons.wikimedia.org/w/index.php?curid=36926283>

		True condition			
		Condition positive	Condition negative	Prevalence = $\frac{\sum \text{Condition positive}}{\text{Total population}}$	Accuracy (ACC) = $\frac{\sum \text{True positive} + \sum \text{True negative}}{2 \cdot \text{Total population}}$
Predicted condition	Predicted condition positive	True positive	False positive, Type I error	Positive predictive value (PPV), Precision = $\frac{\sum \text{True positive}}{\sum \text{Predicted condition positive}}$	False discovery rate (FDR) = $\frac{\sum \text{False positive}}{\sum \text{Predicted condition positive}}$
	Predicted condition negative	False negative, Type II error	True negative	False omission rate (FOR) = $\frac{\sum \text{False negative}}{\sum \text{Condition negative}}$	Negative predictive value (NPV) = $\frac{\sum \text{True negative}}{\sum \text{Predicted condition negative}}$
	True positive rate (TPR), Recall, Sensitivity, probability of detection, Power = $\frac{\sum \text{True positive}}{\sum \text{Condition positive}}$	False positive rate (FPR), Fall-out, probability of false alarm = $\frac{\sum \text{False positive}}{\sum \text{Condition negative}}$	Positive likelihood ratio (LR+) = $\frac{\text{TPR}}{\text{FPR}}$	Diagnostic odds ratio (DOR) = $\frac{\text{LR}^+}{\text{LR}^-}$	$F_1$ score = $\frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$
	False negative rate (FNR), Miss rate = $\frac{\sum \text{False negative}}{\sum \text{Condition positive}}$	Specificity (SPE), Specificity, True negative rate (TNR) = $\frac{\sum \text{True negative}}{\sum \text{Condition negative}}$	Negative likelihood ratio (LR-) = $\frac{\text{FNR}}{\text{TNR}}$		

source: [https://en.wikipedia.org/wiki/Confusion\\_matrix](https://en.wikipedia.org/wiki/Confusion_matrix)

77

There are many more metrics which you can derive from the confusion matrix. Wikipedia provides a helpful table, in case you ever come across them. For most purposes, **precision**, **recall**, **accuracy** and **balanced accuracy** are sufficient.

Note that some terms, like recall, go by *many* different names.

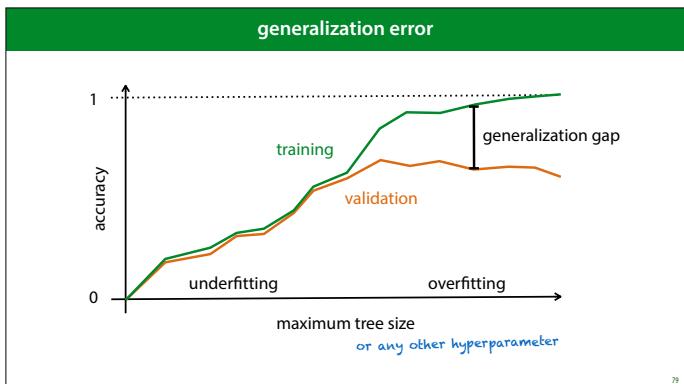
which dataset?	
test accuracy	final test of model performance
validation accuracy	to choose hyperparameters
training accuracy	???

78

All of these metrics can be applied to different datasets. When we compute (say) accuracy on the test set, we talk about **test accuracy**. This is computed—only once—at the very end of our project, to show that our conclusions are true.

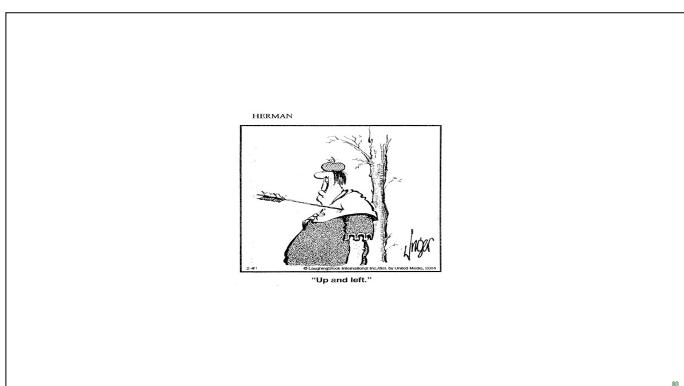
When we compute it on the validation set we call it **validation accuracy**. We compute this to help us choose good hyperparameters.

And, predictably, when we compute it on the training data, we call it **training accuracy**. Remember that in the first lecture I said, emphatically, that you should never judge your model on how it performs on the training set. Why then, would you ever want to compute the training accuracy (or any other metrics on the training data)?

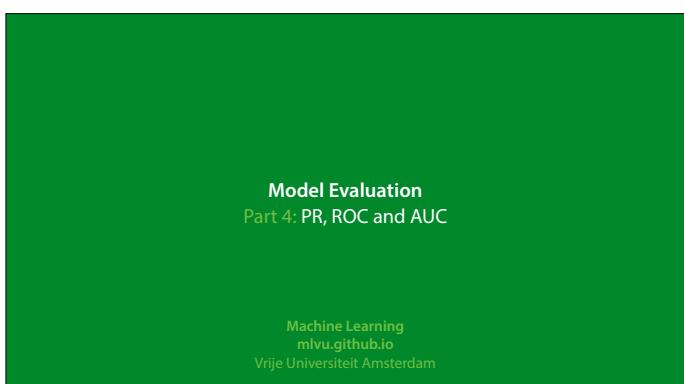


The answer is that the difference between your **validation accuracy** and your **training accuracy**, will tell you whether or not your model is overfitting (matching the data too well) or underfitting (not matching the data well enough).

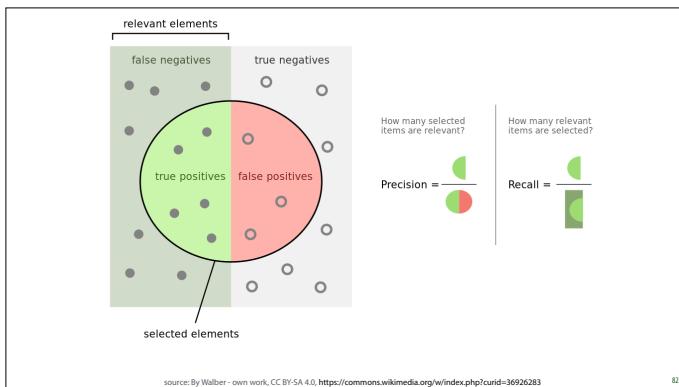
The difference between the training and validation sets is called the **generalization gap**. As in, it's the amount of performance that won't generalize to data that isn't your training data.



source: Jim Unger, Herman.



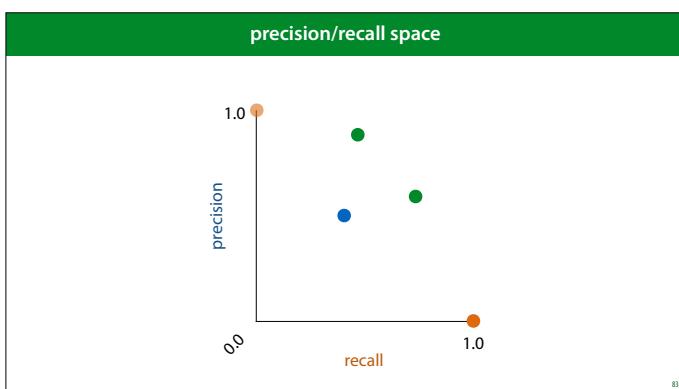
|section|PR, ROC and AUC|  
|video|<https://surfdrive.surf.nl/files/index.php/s/p6Atngg3Q5AzMiz/download>|



Let's return to the metrics of precision and recall. We often have to make a tradeoff between high precision and high recall. We can boost our recall by calling more things positive. The drawback is that our precision will go down if this means including more negatives among the things we call positive. We can boost our precision by calling fewer things positive, which will hurt our recall.

How exactly we make the tradeoff depends on our cost imbalance, and our class imbalance. To help us investigate, we can plot the precision and recall we get from different classifiers.

source: By Walber - own work, CC BY-SA 4.0, <https://commons.wikimedia.org/w/index.php?curid=36926283>



The **points in the corners** represent our most extreme options. We can easily get a 1.0 recall by calling everything positive (ensuring that all true positives are among the selected elements). We can get a very likely 1.0 precision by calling only the instance we're most sure about positive. If we're wrong we get a precision of 0, but if we're right we get 1.0.

Whether we prefer the left or the right **green classifier** depends on our preferences. However, whatever our preference, we should always prefer either **green classifier** to the **blue classifier** since both have better precision *and* recall than the blue classifier.

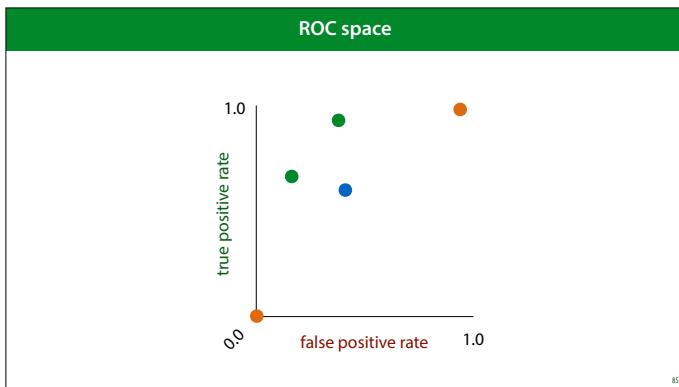
		predicted		accuracy
		pos	neg	$(TP + TN) / \text{total}$
actual	pos	TP	FN	true positive rate $TP / (TP + FN)$ TP/actual pos
	neg	FP	TN	
				false positive rate $FP / (FP + TN)$ FP/actual neg

Another pair of metrics that provides this kind of tradeoff are the true positive and false negative rate.

**true positive rate:** what proportion of the actual positives did we get *right*. The higher the better. I.e. How many of the people with cancer did we detect.

**false positive rate:** what proportion of the actual negatives did we get *wrong* (by labelling them as positives). The lower the better. I.e. How many healthy people did we *diagnose* with cancer.

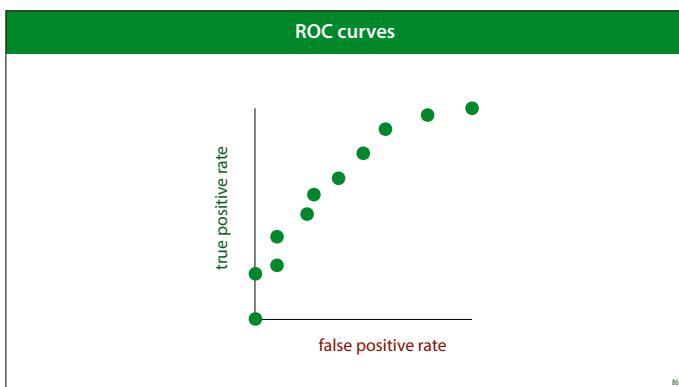
The



We want to get the TPR as high as possible, and the FPR as low as possible. That means the TPR/FPR space has the best classifier in the top left corner. This space is called ROC space.

Again, the orange points are the extremes, and easy to achieve.

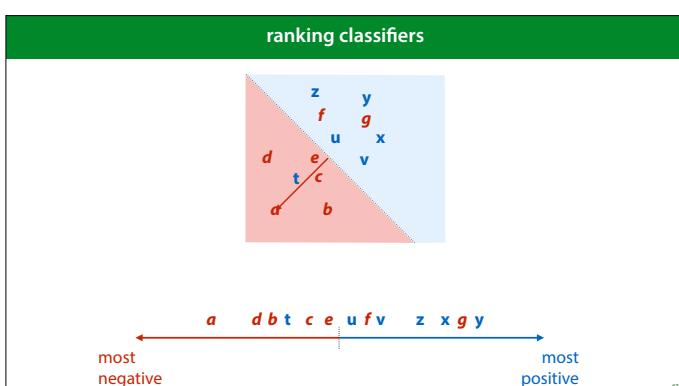
*ROC stands for receiver-operating characteristic. Like so many names in machine learning, this is a historical artifact, a leftover from its invention in WWII to improve the detection of Japanese aircraft from radar signals.*



So far we've thought of FPR/TPR and precision/recall as a way to analyze a given set of models.

However, what if we had a *single* classifier, but we could control how eager it was to call things **positive**? If we made it entirely timid, it would classify nothing as positive and start in the bottom left corner. As it grew more brave, it would start classifying some things as positive, but only if it was really sure, and its true positive rate would go up. If we made it even more daring, it would start getting some things wrong and both the TPR and the FPR would increase. Finally, it would end up classifying everything as positive, and end up on the top right corner.

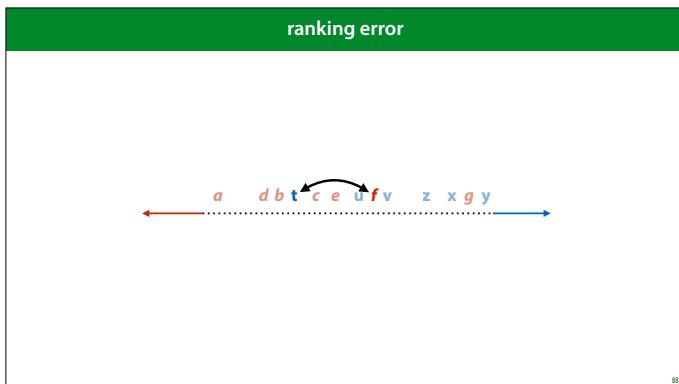
The curve this classifier would trace out, would give us an indication of its performance, *independent* of how brave or how timid we make it. How can we build such a classifier?



We can achieve this by turning a regular classifier into a **ranking classifier** (also known as a **scoring classifier**). A ranking classifier doesn't just provide classes, it also gives a *score* of **how negative** or **how positive** a point is. We can use this to rank the points from most negative to most positive.

How to do this depends on the type of model we use. Here's how to do it for a linear classifier. We simply measure the distance to the decision boundary. We can now scale our classifier from timid to bold by moving the decision boundary from left to right.

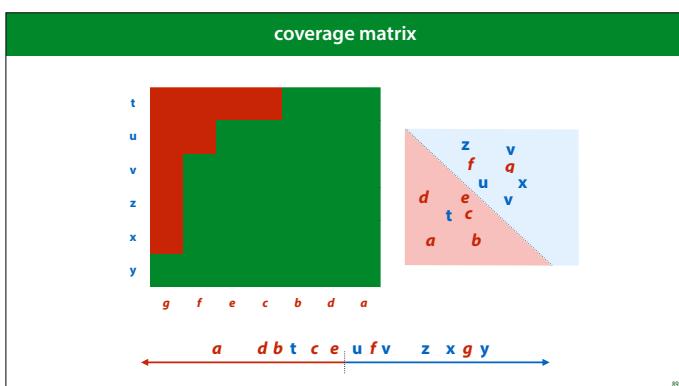
After we have a ranking, we can scale the eagerness of the classifier to make things positive. by moving the threshold (the dotted line) from left to right, the classifier becomes more eager to call things negative. This allows us to trade off the true positive rate and the false positive rate.



Now, we can't test a ranking on our test data, because we don't know what the correct ranking is. We don't get a correct ranking, just a correct *labeling*.

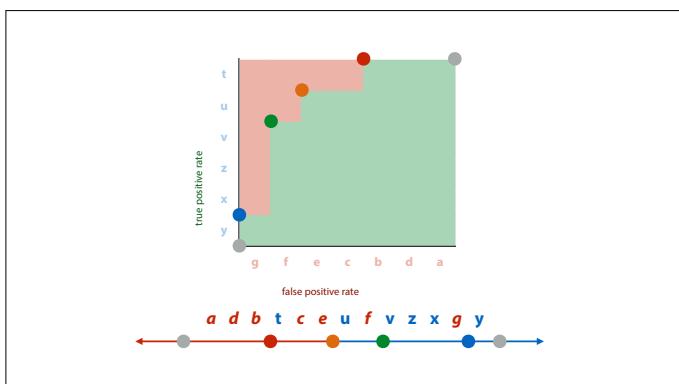
However, we can indicate for specific pairs that they are ranked the wrong way around: all pairs of different labels. For instance, **t** and **f** form a ranking error: **t** is ranked as **more negative** than **f**, even though **t** is **positive** and **f** is **negative**.

Note: a ranking error is a *pair* of instances that is ranked the wrong way around. A single instance can be part of multiple ranking errors.



We can make a big matrix of all the pairs for which we know how they should be ranked: **negative** points on the horizontal axis, **positive** on the vertical. The more sure we are that a point is positive, the closer we put it to the bottom left corner. This is called a **coverage matrix**. We color a cell **green** if the corresponding points are ranked the right way round, and **red** if they are ranked the wrong way round.

Note that the proportion of this table that is **red**, is the probability of making a ranking error.



The coverage matrix shows us exactly what happens to the true positive rate and the false positive rate if we move the threshold from the right to the left. We get exactly the kind of behaviour we talked about earlier. We move from the all-positive classifier step by step to the all-negative classifier.

### warning: exam question

type: find a ranking

We have the following training set:

	$x_1$	$x_2$	label
a	0	1	Neg
b	2	2	Neg
c	1	4	Neg
d	2	5	Neg
e	3	6	Pos
f	6	8	Pos
g	5	3	Pos
h	8	7	Pos

For the following questions, it helps to draw the data and the classification boundary in feature space.

We use a linear classifier defined by

$$c(x_1, x_2) = \begin{cases} \text{Pos} & \text{if } 0 \cdot x_1 + x_2 - 2 > 0 \\ \text{Neg} & \text{otherwise.} \end{cases}$$

13. If we turn  $c$  into a *ranking* classifier, how does it rank the points, from most Negative to most Positive?

A a b c d e f g h

B b a c d e f g h

C c b a d e f g h

D d c b a e f g h

E e d c b a f g h

F f e d c b a g h

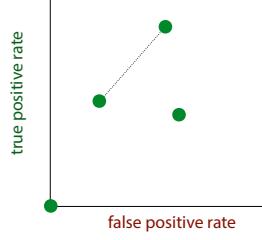
G g f e d c b a h

H h g f e d c b a

This is one of the question types on the exam. People very often make mistakes in this question, so make sure you understand what a ranking error is. **It's not a misclassified example.** It's a property of a pair of examples.

There are more details in the third homework.

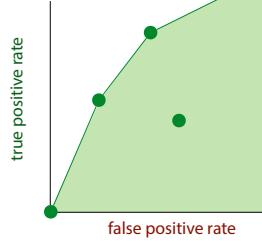
### achievable rates



If we draw a line between two classifiers we know we can create, we can also create a classifier for every point on that line simply by picking the output of one of the classifiers at random. If we pick with 50/50 probability we end up precisely halfway between the two.

If we vary the probability we can get closer to either classifier.

### area under the curve

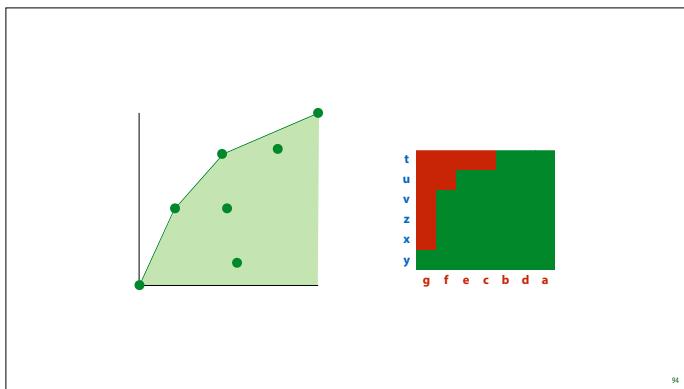


This means we can create any classifier in this green area, called the convex hull of the set of green dots. This is called the area under the (ROC) curve.

The AUC is a good indication of the quality of the classifier.

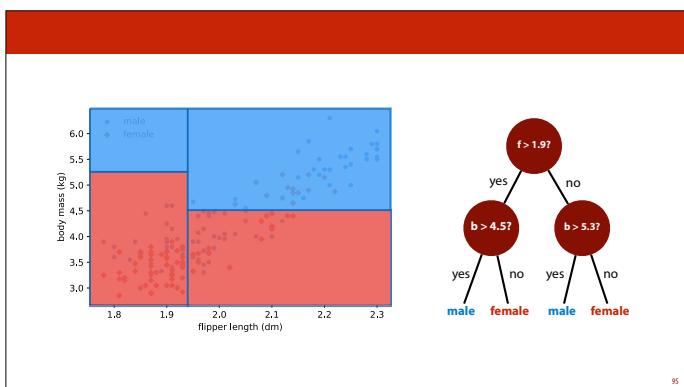
*Every point in the green shaded area is a classifier we now how to make, so the bigger this area, the more classifiers we can make (although the ones on the boundary are the only ones we'd ever want to use).*

If we have no idea of how we want to make the tradeoff between the TPR and the FPR, the AUC may be a good way to compare classifiers in general.



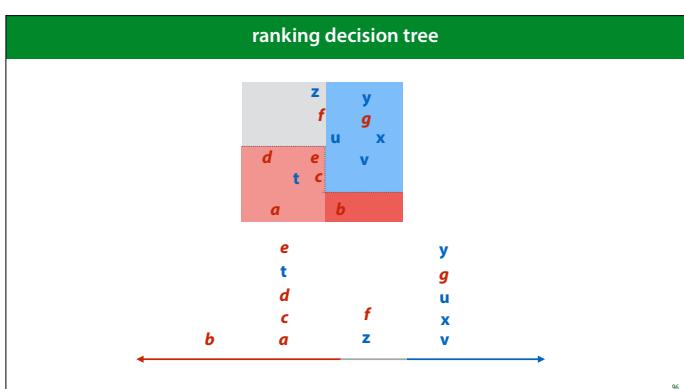
As we saw before: normalizing the coverage matrix gives us the ROC space (barring some small differences that disappear for large datasets). The area under the ROC curve is an estimate of the green proportion of the coverage matrix. This gives us a good way to interpret the AUC.

**The AUC (in ROC space) is an estimate of the probability that a ranking classifier puts a randomly drawn pair of positive and negative examples in the correct order.**



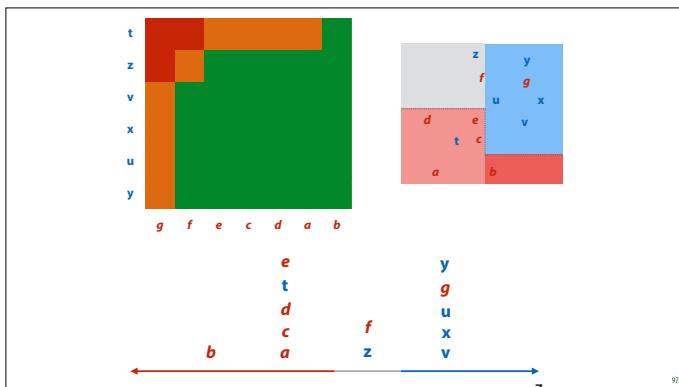
Let's look at how this works for another type of classifier. To reiterate, **how we get a ranking from a classifier depends entirely on the model class.**

The decision tree is an example of a *partitioning* classifier. It splits the feature space into partitions, and assigns each partition, also known as a **segment**, a class. All instances in the segment get the same class.



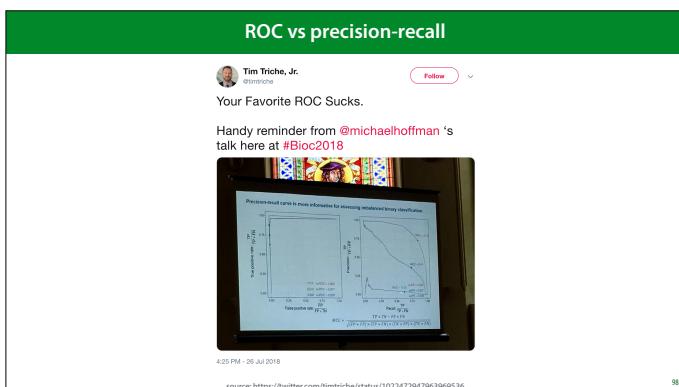
In this example we have an instance space that has been split into four segments by a decision tree. We rank the segments by the proportion of positive points. We then put all points in one region on the same level in the ranking.

In this example, **b** is more **negative** than **a**, because **b**'s segment contains only negative examples, whereas **a**'s segment contains a mix of **positive** and **negative** examples.



This means that for some pairs (like **f,z**), the classifier ranks them as "the same". We'll color these cells **orange** in the coverage matrix.

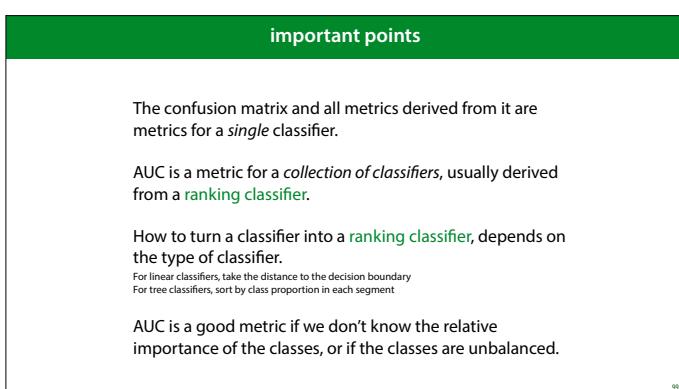
For large datasets, these regions will not contribute much to the total area under the curve.



An alternative to the ROC is the **precision/recall curve**. It works in exactly the same way, but has precision and recall on the axes.

As you can see in this tweet, in many settings the PR curve can be much more informative, especially when you're a plotting the curves. Practically, it's little effort to just plot both, and judge which one is more informative.

ROC has the benefit of an intuitive interpretation for the AUC (the probability of ordering a random pair the right way round). I haven't yet found a similar interpretation for the PR-AUC.



To interpret the AUC, you should know not just what classifier was used, but how it was made into a collection of classifiers. You should also know whether it's the area under an ROC curve, or a precision/recall curve.

## setting the threshold

Show the user the ROC/PR curve, let them choose  
This can be difficult to do accurately.

Estimate cost of misclassifications. Factor into the loss  
function. Minimize the [expected cost](#).

In sklearn, this is done by setting `class weights`. If a false negative costs as much as three false  
positives, we set the positive weight to 3 and the negative weight to 1.

To put a classifier into production, a ranking may not be enough. Sometimes, you just need to produce a single answer. In that case, you can still use the ROC and PR curves to tune your hyperparameters and choose your model, but ultimately, you'll need to choose a threshold as well: the point at which you decide to call something a positive.

This is more of a software development issue than a scientific choice. Often, you have to look carefully at the curves, perhaps together with the end users, to make a decision.

The second approach works best with probabilistic classifiers, which we'll discuss next lecture.

## recap so far

split your data into [train/val/test](#)

accuracy is great, unless you have [class imbalance](#) or [cost imbalance](#)

if you do, look at your:  
confusion matrix  
precision/recall space  
ROC space

if you need a single number: try ROC-AUC or PR-AUC

101